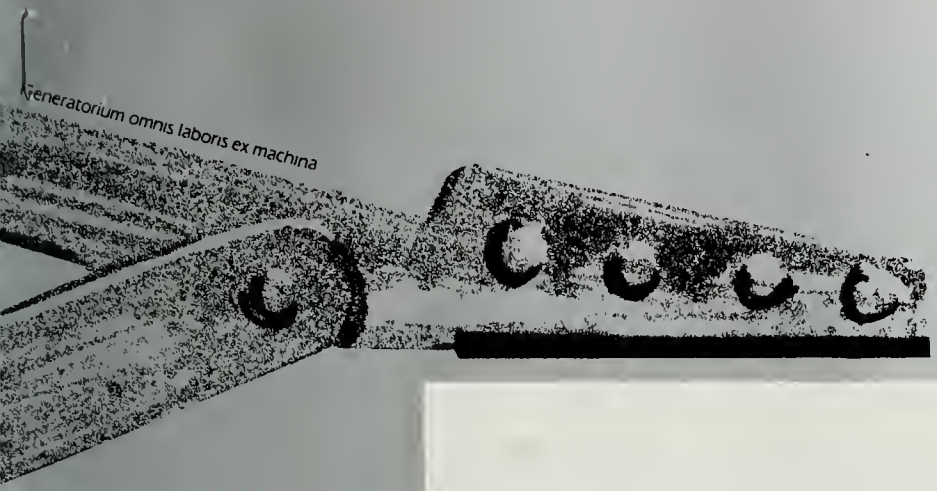
Robotics Research Technical Report

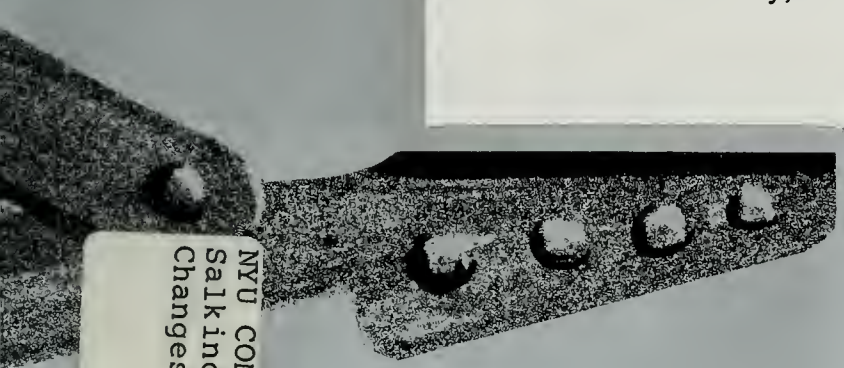


Changes in NRTX

by

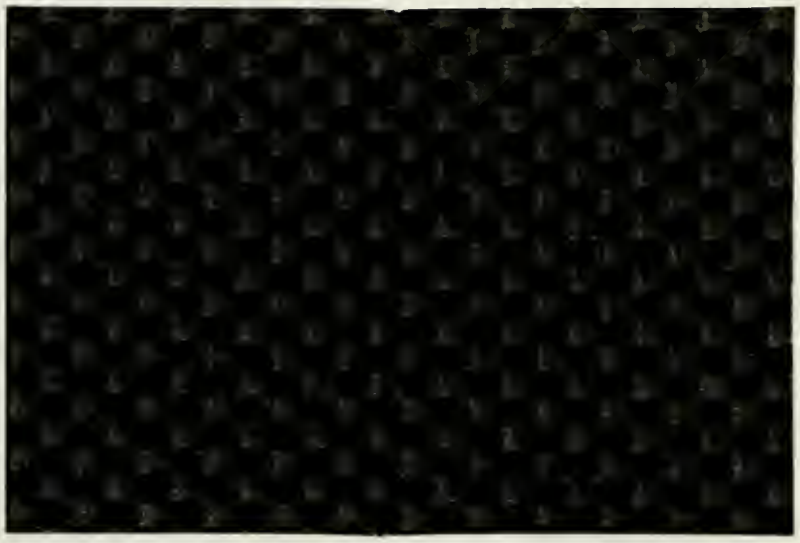
Lou Salkind

Technical Report No. 306
Robotics Report No. 114
July, 1987



New York University
nt Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012



Changes in NRTX

by

Lou Salkind

Technical Report No. 306
Robotics Report No. 114
July, 1987

New York University
Dept. of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, New York 10012

Work on this paper has been supported by Office of Naval Research Grant N00014-82-K-0381, National Science Foundation CER Grant DCR-83-20085, and by grants from the Digital Equipment Corporation and the IBM Corporation.

Changes in NRTX

Lou Salkind
Courant Institute

June 3, 1987

1 Introduction

NRTX is a multi-tasking, real-time operating system developed at AT&T Bell Laboratories for use with Motorola 68000 based systems. Typically, NRTX application programs are written in the C language on a standard UNIX[1] time-sharing system, cross-compiled, and then downloaded onto the target NRTX system. The NRTX system call interface is similar to that provided by a UNIX Version 7 system, with various modifications and additions for real time applications. It is further described in [1].

We decided to use the NRTX system for the development of a force-position controller for the PUMA 560 Robot arm. NRTX seemed like the natural choice for the operating system of the controller, since it supported a powerful microprocessor, it had a familiar development environment, and it was already successfully being used in a related lab project, the force control system of a Planar Four Finger Manipulator [2].

Unfortunately, the NRTX system as originally provided could not reliably support the PUMA force control applications. The system could not meet hard real-time constraints, and limitations in the operating system made sophisticated applications difficult to program. It soon became clear that the operating system would have to be modified if more sophisticated applications were to be successfully run.

During the last year, the NRTX system has been overhauled in an attempt to overcome the system's key limitations. This paper discusses the changes that have been made to the NRTX system to support the PUMA control system. The changes can roughly be grouped into two major categories: changes to the operating system kernel, and changes to the program support libraries and utilities.

[1] UNIX is a Trademark of AT&T Bell Laboratories

2 Controller Architecture

To better understand the motivation for the changes made to NRTX, we digress briefly to describe the design of the force-position controller.

The main goal of the force-position controller is to compute new robot set-points, given the current position of the robot arm and the forces acting upon the tool tip. This force-position controller design is adequate for our needs, since we are more concerned with quasi-static planning issues than with dynamic control of the robot arm.

The controller does not perform the lowest level of control; this is left to other processors interfaced to the controller. For example, the PUMA arm controller performs the servo joint control and inverse kinematics calculations, while a force sensing wrist converts strain gauge readings into Cartesian force and torque values with respect to a given reference frame. This design relieves the force-position controller of the most time critical control computations, which are better done on dedicated microprocessors anyway. However there is a tradeoff, since the controller's communications facilities are now heavily taxed. This is because the controller must communicate with both the PUMA arm and the wrist at very high speeds.

3 Kernel Changes

The kernel implements the multi-tasking environment and allows the sharing of resources among the various processes. As described below, changes have been made in most every kernel facility.

3.1 TTY Drivers

Tty lines (RS-232 serial lines) are used by the force controller for many functions, including communicating with both users and other devices, so it is important that the system provide both an efficient and flexible way of programming these devices. When tty lines are used for communicating with users, the system should provide convenient command line editing facilities, flexible input and output translation, etc. When communicating with other devices, the system should impose very low overhead and latency, while still maintaining a simple system interface.

The old NRTX tty driver was based on the UNIX V7 system, which turned out to be both too slow and limited to support the desired applications. Therefore the tty interface was upgraded to support the 4.3 BSD tty line discipline [3]. Although this driver is slightly larger than the V7 version, it provides a much better user interface and is also much more efficient.

Even with the new Berkeley tty driver, machine to machine communications over a serial line still imposed a high overhead on application programs. This

is because when a program wanted to read binary data (i.e., all eight bits), a read system call would return as soon as any characters were received, no matter how many characters had been requested. This tended to cause needless context switching between processes. Although various subterfuges can mitigate the problem somewhat, these work-arounds introduced other problems in our applications.

Fortunately, extensions to the UNIX tty driver interface can be provided in an upwards compatible fashion through the use of line disciplines. Such disciplines are compiled into the kernel at system generation time, and specify an alternative way of processing characters that pass between a serial device and an application.

We wrote two line disciplines that are particularly well suited for high speed machine to machine communication:

block discipline In block mode, input and output characters are processed in exactly the same way as the default tty driver discipline. The difference between the two modes is that in block mode, the read system call will return only when a specified number of characters have been read, or optionally, when a character in a specified termination mask has been read. This discipline is generally useful for many applications, and could also be easily included in a standard UNIX system. In the force controller this discipline is used for reading force and torque values from the wrist.

alter discipline This discipline is used for communicating with the PUMA arm controller using the VAL-II motion modification input line. This line discipline understands the byte stuffing and flow control protocols used for communication with the PUMA controller. It is also a one of a kind discipline, in the sense that it probably can not be used for communicating with any other device.

These line disciplines allow much more efficient use of the system; this is essential, since the force controller typically processes about 2500 characters per second.

3.2 Distributed Communications

The communications support originally provided in NRTX turned out to be inadequate. Although an Ethernet controller was supported, only one process at a time could use the device. In the force-controller software design, several processes had to communicate concurrently over the Ethernet. Therefore, it became clear that some sort of scheme would have to be implemented to multiplex many communications streams onto a single device.

We decided that such a mechanism best resided in the kernel, and inspired once again by the Berkeley UNIX system, introduced a socket mechanism for

communications. The socket model is open-ended in that it can potentially support an arbitrary number of protocols running over different network interfaces.

Each socket protocol is supported inside the kernel by a separate module, while each network interface is supported by its own device driver. There are well defined interfaces between the driver level, protocol level, and the application level. To date the following protocols have been implemented:

UDP/IP The UDP/IP protocol is a datagram oriented protocol and is supported on a wide range of machines, including all Berkeley UNIX systems. Thus NRTX can communicate with all our time-sharing systems in the lab. This is not a full implementation of the IP protocol: no fragmentation or re-assembly of packets is performed, and ICMP is not supported. However, there are facilities that provide static routing of packets.

ARP ARP (address resolution protocol) is supported. This protocol resides entirely in the kernel, and is needed for locating host protocol addresses on an Ethernet cable. Logically, it resides in between the driver and protocol level.

Raw Packet Protocol The raw packet protocol allows the user to send arbitrary packet types on the Ethernet (or another network device).

Currently supported interfaces are a 3-Com Ethernet controller and a loopback driver. The Ethernet controller is used for all inter-machine communications, while the loopback driver is used for intra-machine communications and testing.

3.3 Message Passing

The original NRTX provided a message passing facility for interprocess communication. Unfortunately, this facility did not turn out to be very useful for force control applications. Among other problems, we found that:

- it was impossible to change message queue sizes
- not enough control was provided over how messages could be handled by the receiver
- it was impossible for one process to manage more than one receiving queue, since queues were allocated on a one per-process basis.
- the message passing facility only worked for processes residing on the same machine, and many of the force control applications needed to communicate with processes on other machines.

These deficiencies forced us to abandon the NRTX message passing scheme. However, we still wanted to use a message passing style of communications

for several controller applications, so we wrote several library packages that implement different message passing schemes.

Currently, two packages exist for message passing. The first package basically replaces the functionality provided by the old NRTX message passing facility, but it is implemented at the application level using shared memory. Various parameters are tunable at either compile or run-time, thus eliminating many of the problems found in the old message passing interface. The second package, which facilitates remote procedure to different hosts, provides a fundamentally different model for communications. Built on top of the UDP socket mechanism, this package provides a simple way of communicating with other hosts in the lab.

By not rigidly specifying the message passing interface in the kernel, we can experiment with different protocols and tailor each message passing implementation to the given application. We can do this with little performance penalty, since there is little duplication of work between the kernel and application code, and almost all data is passed by reference between the two levels. At some point in the future, however, we may provide a socket type that performs reliable message passing, thereby moving more of the message passing functionality into the kernel.

3.4 Synchronization Facilities

NRTX originally provided only binary semaphores and the UNIX sleep-wakeup primitives for process synchronization. We found it useful to also provide support for counting semaphores, which made it easier to write consumer-producer type applications.

Fortunately, counting semaphores could be added to the system while incurring almost no additional overhead. At the same time, the semaphore interface was cleaned up to provide temporary and globally accessible semaphores, allow for non-blocking operations on semaphores, and to initialize semaphores to arbitrary values.

3.5 Timer Facilities

NRTX provides a variety of timer facilities, including the ability to read and write the time of day clock, suspend execution of a process for a given time interval, or interrupt execution of a process after a given time interval. We feel this is an adequate set of primitives.

Unfortunately, the timer resolution in the original version of NRTX is too coarse for control purposes. For example, the time of day could only be read to an accuracy determined by the underlying line clock interrupt frequency, and in no case with an accuracy greater than 1 millisecond. Also, the interval timers could not be specified in units smaller than .01 seconds.

We have modified the system call interface so that all time values are specified in units of microseconds, which avoids rounding problems. In addition, the system call that reads the time of day clock has been modified to return values accurate to the microsecond, regardless of the underlying clock interrupt frequency. The interval timers, however, are still rounded up internally to the appropriate number of clock ticks, so they are still only as accurate as the the underlying clock interrupt frequency. So far this has been adequate for our needs.

3.6 Non-Blocking System Calls

The ability to execute a system call conditionally, based on whether or not the process would block, was deemed important for time-critical operations. To provide this ability, several parts of the system had to be modified, including the tty disciplines, the wait system call, and the semaphore interface.

For the tty disciplines (as well as for other character oriented devices), the user can now specify a non-blocking mode for the device descriptor. In non-blocking mode, an operation on the descriptor will return with a special error code if the system call would block indefinitely.

The wait system call has been changed to accept a flag argument which specifies a non-blocking mode. If there is no child process that has terminated, the system call will return immediately. This allows the process server, for example, to continue to serve requests in a real-time fashion.

Finally in the semaphore interface, we added a system call to atomically test a semaphore value and decrement it by a specified amount, provided the value would not become negative. If the value would become negative, the semaphore value remains unchanged and the call returns immediately. Thus this call simulates a general test and set operation.

3.7 Memory Management

One of the major design decisions in the original NRTX was to not use any memory management scheme and to have all processes run in the same address space. There were two major benefits of doing this. First, since no memory management was used, obviously no memory management unit would be required. Therefore, not only would the required hardware be less expensive, but the system would be simpler and easier to port to other boards. Secondly, without memory management, there would be less process context required, and so there would be less overhead in context switching and in the creation and destruction of processes.

Unfortunately, we found that the old scheme was just too dangerous and limited. The system was dangerous because one failing process could easily corrupt the system kernel or another process in a nearly untraceable fashion. The system was too limited since the only operations a process could perform

were allocating and freeing physical memory. There was no general way of sharing read-only text and data areas among many different processes.

In order to provide both protection and flexibility in our system, we decided to take advantage of the hardware memory management unit on our system board and provide a limited virtual memory system without demand paging. We felt this was a reasonable decision, since many one-board systems for 32 bit micros now support both memory management and virtual memory. The main challenge was to do this in such a way that minimal overhead and complexity would be added to the system.

The new virtual memory interface is based on a segment model of memory. Each segment is a sub-interval of the total virtual address space available to a process and has various attributes, among which are:

size The segment size is the range of virtual addresses in this segment, and is specified at the time the segment is created.

permissions A segment can be read-only, read-write, invalid, etc. Thus the operating system can guarantee that pure text segments and read-only data will not be corrupted.

name Each segment has a unique name, and other processes can attach to a segment by specifying its name. Therefore segments can be easily shared among several processes.

mapping Virtual addresses in segments can be mapped onto either physical memory or i/o address space. Thus segments allow a process to effectively control devices without performing system calls that might incur intolerable overhead.

references The system keeps track of how many processes have attached to the segment. When the reference count goes to zero, the system can return all the segment's resources back to the system. It is also possible to create segments that can exist even when the reference count is zero; this feature can be used for dynamically linking to library text and data modules.

All these attributes can be specified or changed using various system calls, so a knowledgeable process can exercise full control over the memory management functions.

The major benefit of the segment model is that it provides a convenient way for processes to manage memory safely while at the same time allowing the system to keep track of things. Since the segment model is relatively simple, it can be implemented on a wide range of memory management units. In fact, the segment model can also be implemented on systems without memory management, although there will be correspondingly less protection and mapping capabilities.

In the current implementation, all processes still share the same virtual address space. Therefore there is no extra overhead imposed by the memory management scheme when switching process context, since page and segment tables need not be adjusted. Of course, since all segments are now mapped in to the process' address space, an errant process can modify any writeable segment in the system. However, in practice, this turns out to be a minor consideration, since all of the unmapped address space is invalidated, and all read-only data is also protected in hardware. Thus most stray references are still detected under the new scheme.

In fact, the only extra overhead incurred when using segments is in carrying out the control operations that attach to a segment or change a segment's attributes. For our applications, such operations occur very infrequently relative to how often a segment is accessed, and such operations are never performed in time critical portions of the code, so this extra overhead is of no consequence to us.

3.8 Process Creation

In the original NRTX system, a process was composed of two writeable blocks of memory, which we call the program area and the stack area. The program area contained program text and statically allocated data, while the stack area was used by the C run-time environment. When creating a new process, one specified both the program and stack areas of the new process, as well as the entry point of the new process. Optionally, a new process could attach to the program area of its parent, and the kernel would keep track of how many processes were using the program area. When the last process using the shared area terminated, the system would return the memory to the free memory pool.

Although this scheme made it fairly easy to create new processes that shared a common data area, there were several annoying problems. For instance, a process could only share the program area of its parent, and it was impossible to share stack segments across different processes. Worse, all segment sizes were fixed for the life of the process, making shared heaps difficult to implement in a reliable manner. Finally, there was no real way to pass arguments to the child process when sharing text and data segments.

By using the new segment scheme and providing a different interface, the new process creation scheme fixes these problems and at the same time provides more functionality. A process is now composed of three segments: a text segment, a data segment, and a stack segment. The attributes and contents of these per-process segments are otherwise undefined, but by convention the text segment is a fixed length read-only block that contains program text, the data segment is a fixed length read-write block that contains statically allocated data, and the stack segment is a read-write virtual address block that will be allocated physical memory on an as-needed basis.

The new system call interface is motivated by the standard UNIX system

calls for changing process contexts. There is now a *fork* system call, which like the corresponding UNIX call, creates a child process that shares the text segment of its parent. However, rather than making a copy of the parent's data and stack segments as in UNIX, the NRTX call shares the parent's data and stack segments by attaching to them. Copying segments was not considered attractive, but not only for the extra overhead required for the copying in process creation; it would also force all processes to reside in different address spaces, implying more context per process.

However, copying segments (as done in UNIX) does have its advantages, because the child has safe access to the environment of the parent at the time of the fork call. In particular, many UNIX programs use this behavior of fork to implicitly pass arguments from parent to child. To get the same functionality in NRTX, we allowed the process to specify the new program counter and stack pointer of the child process. Then in order to pass per-process arguments to the child, the parent process copies its arguments to a new run-time stack and then sets the child stack pointer to reference those arguments.

To change the execution context of a NRTX process, one now uses the *exec* call. Its effect is similar to its UNIX counterpart, although it has a slightly different syntax, reflecting the fact that there is no filesystem abstraction supported by the kernel. In the NRTX call, one specifies a new text, data, and stack segment (as opposed to a UNIX filename). Upon successful completion of the call, the current process is effectively overlayed by the new segments.

Note that the original NRTX also provided an exec call, but with totally different semantics. The old version attempted to combine the functionality of the current fork and exec calls. However, we have found the new interface, to be more useful for allowing one process to handcraft the environment of another process, and this ability is used to great advantage in the process loader. In addition, the new NRTX interface is more in keeping with the standard UNIX interface.

3.9 System Configuration

In the original NRTX system, configuration dependent parameters were spread out over many files, and building a system for a particular application and hardware configuration was tedious and error prone. To fix this problem, we reorganized the kernel source code structure and made many small changes. The net result is that now all configuration dependent information resides in just three files, there is less information to worry about, and the system is easier to configure. To build a system, one first edits these files and then uses the make command to build the system from common sources (which reside in other directories). Therefore, it is quite easy to support multiple systems and configurations simultaneously.

The code reorganization also had another benefit, namely, that the system is now easier to modify and develop. In particular, this has allowed us to quickly

add support for new hardware. To date, the following hardware support exists:

Processors The system supports all current members of the Motorola 68000 processor family, including the 68000, 68010, and 68020 CPUs. The original system only supported the 68000 CPU.

Processor Boards The following Pacific Microsystems boards are now supported: M68DF, M68DS, M68DL, M682, and PV682 (this includes both Multibus and VME based systems). The major hardware components of the processor board include the CPU, serial line controller, memory management unit, line clock, bus interface, and interrupt controller.

Floating Point The kernel can optionally support the M68881 floating point chip or the Sky board (the old system only provided minimal support for the Sky board). The system also preserves floating point registers on context switch (the old system didn't). Fortunately, processes that don't use floating point suffer little performance penalty.

Devices The MTI terminal multiplexor is now supported. The MTI is a DMA device that provides DMA on both input and output. In order to efficiently support this device, we added several generic facilities needed by all DMA devices (e.g., mapping memory addresses to bus addresses).

In general, we have found the system to be easy to port to the different CPU, I/O bus, and memory management unit types supported by Pacific Microsystems M68000 boards. We would expect this to be equally true for other manufacturer's 68000 based systems.

3.10 Signal Handling

The original NRTX system provided a signal handling mechanism that was essentially identical to that found in the original UNIX Version 7 system. Basically, a process could arrange for a signal handler to be called upon reception of an exceptional event such as a keyboard interrupt or floating point exception. An uncaught signal would terminate a process.

Unfortunately, the original UNIX signal mechanism suffered from various well-known problems. In particular, there was no way to prevent a race condition whereby a second signal could be delivered to a process before it the first signal was handled. This would cause the process to terminate unexpectedly.

Since these signal problems were fixed in the Berkeley UNIX implementation, we decided to port this code to NRTX. In the end, using the "new" Berkeley signal interface not only made the NRTX signal handling more reliable, but more functional as well. For instance, more signals are now supported. This includes SIGCHLD, which informs a parent when the child has terminated, and SIGIO, which alerts a process when I/O can be performed on a descriptor. The new signal mechanism also provides the signal handler with more information so that the complete state of the process at the time of the signal can be recovered.

3.11 Scheduler Changes

For our applications, we need a pre-emptive scheduler. Namely, when a high priority process becomes ready, it has to be run as soon as possible. This is especially critical for the interface to the PUMA robot arm controller. In this case, approximately 20 transmitter interrupts have to be processed with a total latency of less than 1 millisecond.

Unfortunately the original implementation could not meet this stringent a time constraint. For one thing, a potentially large amount of work could be performed at interrupt level. Since a process can not run until the interrupt has been serviced, it was possible for a ready process to be locked out for too long a period of time. To fix this, we reorganized the code where possible to avoid tying up the processor at interrupt level. The basic idea is to just set a flag at interrupt level and perform the work after returning from the interrupt.

There was also another situation where a newly readied process could be unduly delayed. This occurred when one process was processing a system call and temporarily raised its priority (this is done often to lock out interrupts in critical sections). If at this moment a higher priority interrupt in turn readied another process, this new process would not be rescheduled until after the first process finished its system call. Our solution was to set a software interrupt whenever a process was rescheduled at hardware interrupt level. This software interrupt takes effect once we return from the hardware interrupt, and the software interrupt handler then calls the process rescheduler.

The current implementation has proved to have a fast enough scheduling response for our needs. However, there is still a possibility that a slew of interrupts could occur in rapid succession before a very high priority process would have a chance to run, since in the current implementation interrupt handlers always take precedence over any process. This may have to be fixed at some point in the future for new applications with more stringent real-time deadlines.

3.12 Miscellaneous

In general, many other parts of the system have been cleaned up, there have been numerous bug fixes, and more hooks have been provided for debugging. Also where possible, we have tried to make the system interfaces look more like the Berkeley UNIX system (as opposed to Version 7 UNIX system), which is the operating system on which we do our program development.

4 Support Software

The NRTX support software encompasses all code that is not kernel-resident in the satellite processor board. It can roughly be broken down into three major components:

C Run-Time Support These modules implement standard facilities such as as I/O, floating point, heap allocation, etc. that are generally assumed to be available. These routines can either be explicitly called by the programmer, as in the case of the standard I/O library, or implicitly called by the compiler, as in the case of floating point support.

System Support These libraries and programs implement a wide range of services that can be used by NRTX applications. Such services include remote filesystem access, interprocess communication, process loading, and kernel bootstrapping.

Program Development Utilities These programs facilitate the writing and debugging of NRTX applications. Some of these programs, such as the cross-compiler, run on the host, while other programs, such as the debugger, run on the NRTX satellite.

For our force controller applications, we had to both modify and add to the original support software.

4.1 C Compiler

We do all our program development on a SUN-3 68020 based system, and therefore we can use the SUN C compiler to cross-compile C programs for the NRTX system. To take full advantage of the SUN compiler's features, we rewrote the front-end program rtxcc (which invokes various passes of the compiler), since the original rtxcc program supported only a subset of the SUN compiler options.

Using the new front-end program, we now have the ability to generate either 68000 or 68020 code, generate floating point code for a variety of floating point units, optionally in-line expand function calls with assembler code, and have the C pre-processor generate dependencies for makefiles.

Adding floating point support in the compiler also required a great deal of run-time support, since the 68000 family has no intrinsic floating point support and we have to support several different floating point co-processors. Such run-time support includes co-processor start up code, math libraries for trigonometric functions, etc. To provide this support, we ported the SUN math libraries and support routines, modifying where necessary the operating system interfaces to work properly with NRTX.

Since the SUN compilers generate a combination of assembler code and calls to various (hidden) run-time routines, by porting the run-time routines, we are potentially able to use any SUN compiler for writing applications; all we have to do is supply the appropriate front-end program that calls the appropriate SUN compiler passes to do the actual code generation. Currently we have added support for both the Fortran and C++ languages.

4.2 C Library Routines

Besides floating point support, a set of standard library routines are usually provided by the C run-time system. Although the C language standard does not attempt to specify in any way the run-time library, sophisticated applications are difficult to write without adequate run-time support. And once we started to write control applications, we found that the NRTX C run-time library was lacking in many ways.

For one thing, the standard NRTX C library was missing many routines that are usually found in C implementations. For example, most C implementations provide the routines malloc and free for dynamic heap memory allocation. However, in the original implementation, these functions were just stub routines that did nothing. We found it too great an inconvenience to code our applications without a heap allocator. Therefore, we wrote an efficient memory allocator (in terms of both space and time) that is fully integrated with the new segment mapping scheme.

Besides the memory allocation routines, other useful routines were found missing from the NRTX C library and added as needed: block copying and comparison routines, queue management routines, etc. Also, facilities such as standard I/O were upgraded where necessary, since the original versions were ported from UNIX library routines that were already many years old and relatively inefficient. In general, all the upgraded routines could simply be copied from the Berkeley UNIX distribution with minimal change, since the NRTX and UNIX system call interface are so similar.

4.3 SRPC library

We found that many of our distributed applications, such as the remote filesystem, used remote procedure calls for communicating. Our goal was to provide a simple package to efficiently implement remote procedure calls. To this end, we developed the SRPC library.

SRPC stands for the Simple Remote Procedure Call library, but the name is somewhat misleading, since the SPRC library does not implement a full RPC protocol. In particular, SRPC does not provide a standard format for data representation, a well-defined interface for generating procedure call stubs, or other facilities typically found in an RPC package [SUN RPC, Xerox Courier]. SRPC only provides a reliable datagram transmission facility that can be efficiently used by remote procedure call packages. It is intended that SRPC should provide a simple enough interface so that clients can reliably send messages to a server, but that the more sophisticated RPC facilities be provided by other packages built on top of the SRPC protocol.

SRPC is a connection based protocol built upon the standard UDP/IP protocol (using the corresponding NRTX socket calls). SRPC provides reliable datagram transmission and connection setup and teardown, while UDP/IP provides

error detection, data encapsulation, multiplexing of communication streams, and packet fragmentation and assembly. The intent is for there to be as little duplication of function as possible between the two layers.

Optionally, SRPC will not compute checksums at the UDP/IP layer. This can result in a large performance improvement, but it should only be done when communicating with a peer process on the same machine, or when using a fairly reliable transmission medium such as ethernet that does its own error checking.

4.4 Remote Filesystem

Many NRTX applications need to perform file I/O. For example, some programs need to read files to obtain configuration data, while other programs need to write tracing data for post-mortem analysis. It is important that NRTX provide a way of reliably performing basic filesystem operations such as read, write, open, close, and create.

In general, we feel that a remote filesystem package is the best way to provide filesystem services for NRTX clients. By having a time-sharing host perform most of the work when performing a file operation, we can greatly simplify the NRTX kernel. Of course, this implies that file operations do not satisfy hard real-time constraints. For our applications, however, this is of no concern.

The original NRTX system modified the standard I/O library to provide remote filesystem support. The library would keep track of which descriptors were for local devices or remote files, and if an operation were to be performed on a remote file, make a remote procedure call. The remote procedure call would then be transmitted over the ethernet and processed by a fileserver on a remote UNIX system. Finally, the fileserver would return the results back to the NRTX client.

Unfortunately, this package suffered from several problems. For one thing, communications errors (such as dropped packets) could lead to a deadlocked state. Secondly, only one process at a time could communicate with the fileserver process. Finally, the remote system call interface was not integrated with the rest of the NRTX system. Since there were special calls for remote file operations, applications had to be aware of whether or not they were dealing with a remote descriptor. Not only is this inelegant, but it runs contrary to our belief that an operating system should provide some measure of device independence.

To fix these problems, we rewrote both the remote filesystem library package that runs on the NRTX system and the filesystem server that runs on the UNIX host. For the NRTX side, we used the SRPC package to perform the remote procedure calls efficiently and reliably. In addition, rather than modifying the standard I/O interface, we modified the NRTX system call interface to check whether it is dealing with a remote file descriptor, and if so, make the appropriate remote procedure call. Checking for remote operations at the system call level effectively makes the remote filesystem transparent to the programmer. On the UNIX host, we modified the fileserver to accept multiple connections

and use the SRPC server protocol.

The net effect of these changes is that the performance, reliability, and functionality of the remote filesystem package has increased.

4.5 Process Loader

Once the NRTX system has been bootstrapped, control is transferred to the process loader, which listens for requests from clients to download programs. In the typical case, the client is just a program on the UNIX development machine that communicates with the process loader.

Since the process loader is used to bootstrap other applications, it is one of the most critical system components. It has therefore undergone quite a bit of change as NRTX has evolved.

One change was to eliminate extraneous functions from the loader. In the original implementation, the loader provided a virtual terminal connection between the two serial ports on the processor board and enabled and disabled the ethernet for other processes. Now these same functions are provided by separate programs, which can be downloaded upon request. Eliminating these dubious functions makes the loader code simpler.

Next, the process loader protocol was changed. The new protocol is built on top of the SRPC protocol and supports additional arguments for crafting the downloaded program's environment. In particular, the client can now independently specify the size of the text, data, and stack segments of the newly created process. The process loader will then use the new segment control operations to make the text segment read-only and to set up a red-zone for the stack segment, which prevents the stack segment from growing too large and clobbering another segment. Another parameter allows the client to specify which device should be used for the standard input, output, and error streams.

Finally, the process loader has been modified to accept process load requests at any time. In the past, the process loader would typically block until the previously downloaded application had finished, unless the background flag had been set. To make this change, we used the new process creation facilities and the non-blocking wait system call.

4.6 Debugging Facilities

The original NRTX debugging facilities were quite poor. About the only thing that could be done to track down errors was to insert print statements in the code. There was no way to single step a program, set breakpoints, get stack backtraces, etc.

To cure this, we ported the ddt program to NRTX. ddt is a 68000-based debugger that allows single stepping, multiple breakpoints, stack traces, symbolic references, memory peeking and poking, etc. Originally developed at Stanford,

ddt has been widely used for debugging many 68000-based systems, including the Apple Macintosh and the NYU Ultracomputer.

We made several modifications to ddt so that it would work properly in the NRTX environment. In particular, ddt had to be changed so that it could read the object file format generated by the Sun (and Berkeley) compilers. In addition, since ddt only supported the 68000 processor, we had to extend the instruction disassembler and trap routines to support the 68010 and 68020 processors as well. Finally, we wrote a utility program that dynamically loads an arbitrary symbol table into ddt's data space. Thus we can use ddt for debugging any executing image, including the kernel itself.

In addition to ddt, one can also trap to the Pacific Microsystem's PROM monitors and use the full range of commands supported by their firmware. This occasionally turns out to be useful for debugging kernel code.

4.7 Miscellaneous

To facilitate program development and maintenance, we have reorganized the directory hierarchy for the support software. There are now many less levels of structure, since we found the old organization to be more of a nuisance than an aid. Also, the machine dependent files have been isolated, making the job of porting the support software to other machines easier.

In addition, we now make extensive use of the UNIX tools for program development and maintenance. All or part of the support software can be built using the make program, and all changes in the support software can now be tracked with the RCS revision control system. Using make and RCS in concert allow us to quickly build the system and then perform regression tests after changing parts of the code.

5 Conclusion

The new NRTX system is more functional, more reliable, and easier to maintain than the original version. In addition, the operating system can now meet the real-time constraints imposed by the various force controller tasks, and these tasks can now be written in a natural way.

Throughout, our philosophy has been to fix the problems we have encountered in the NRTX system rather than trying to work around them. At the same time, however, we have tried to keep to the basic tenets of the original NRTX system, namely, that the system should provide a UNIX-like interface with general purpose real-time facilities.

5.1 Future Work

There are a few more things that should be modified in the kernel. First, the operating system should use both the user and kernel modes of the 68000

(the current implementation uses only kernel mode). This will afford us much more protection, as well as remove several patches in the kernel code, since the application stack can be kept separate from the kernel stack for each process.

We would also like to fix the scheduler up a bit more to avoid latency problems. We should be able to specify the exact time at which timer events occur (there is currently too coarse a quantum), and we have the ability to dispatch processes at interrupt level.

Finally, we would like to add a message passing discipline to the kernel. This would result in both a space and time savings, since several tasks already are use message passing for communication. In addition, a kernel message passing mechanism would allow the operating system to be extended to multiprocessors in a natural way, without having to significantly change the underlying implementation.

At the application level, we hope to write some new tasks in C++, now that the appropriate level of language support is available.

5.2 Acknowledgments

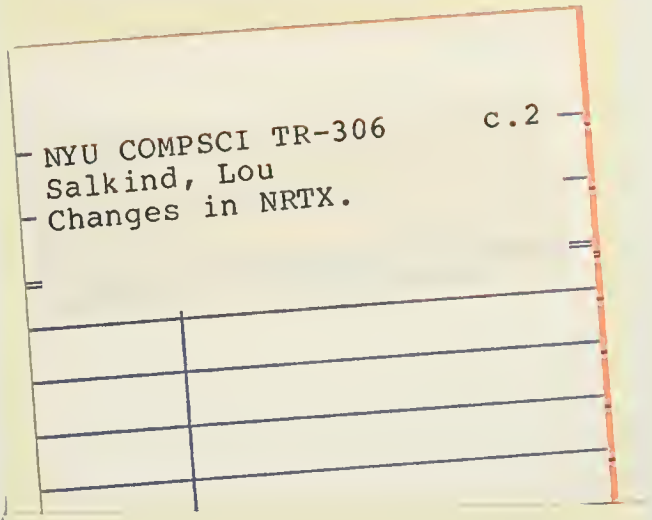
Thanks to David Kapilow, who wrote the original NRTX system, and to AT&T Bell Laboratories, who provided the system to us. Also, thanks to Jim Fehlinger for adding the initial PROM monitor support.

References

[1] D. A. Kapilow, *Real-Time Programming in a UNIX Environment*, Proceedings of the Symposium on Factory Automation and Robotics, Courant Institute of Mathematical Sciences, New York University, September 1985

[2] J. Fehlinger, *An Experimenter's Guide to the Four-Finger Manipulator*, New York University Robotics Report No. 102, February 1987

[3] *UNIX Programmer's Manual*, 4.3 Berkeley Software Distribution, University of California Berkeley, 1986


NYU COMPSCI TR-306 c.2
Salkind, Lou
Changes in NRTX.